CS131 HW1

greeting = "Hello, world!"

1.

-- if length x > length y

Solution:

```
largest x y =
   if length x >= length y then x
                           else y
```

2.

{-

Barey's code does not have () around the num+1 and num-1. Thus, the operation num+1 and num-1 will never execute because Haskell is strict about
the syntax for functions and having the operations within parameters being surroudned by parentheses to ensure it occurs before
it is passed. This means that the function will never reach the base case since it is never incrementing num+1 or decrementing num-1.
To fix the code we would need to add parentheses around it to ensure the functions parameters are being incremented or decremented to reach the base case of 0.
-}

Correction:

```
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
   | num < 0 = (-1) + reflect (num+1)
   | num > 0 = 1 + reflect (num-1)
```

3.

A.

-- needed to place a bounds on the range for x otherwise it will execute unlimitedly

Solution:

```
all_factors :: Integer -> [Integer]
all_factors y = [ x | x <- [1..y], y `mod` x == 0]
```

B.

Solution:

```
perfect_numbers :: [Integer]
perfect_numbers = [ z | z <- [1..], sum (init (all_factors z))== z]
```

4.

```
{-
Python implementation
def is_even(n):
    if n == 0:
        return True
    else:
        return is_odd(n-1)


def is_odd(n):
    if n == 0:
        return False
    else:
        return is_even(n-1)

-}
```

Solution: for IF Statements

```
-- if statements
is_even :: Integer -> Bool
is_even n =
   if n == 0 then True
           else is_odd (n-1)

is_odd :: Integer -> Bool
is_odd n =
   if n == 0 then False
           else is_even (n-1)
```

Solution: for guard Statements

```
-- guard statements
is_even :: Integer -> Bool
is_even n
   | n == 0 = True
   | otherwise = is_odd (n-1)

is_odd :: Integer -> Bool
is_odd n
   | n == 0 = False
   | otherwise = is_even (n-1)
```

Solution: for pattern matching

```
-- Pattern matching
is_even :: Integer -> Bool
is_even 0 = True
is_even n = is_odd (n-1)

is_odd :: Integer -> Bool
is_odd 0 = False
is_odd n = is_even (n-1)
```

5.

THOUGHT PROCESS

{-

Naive: Greedy Approach

If I were to implement this in python, I would use two pointers, one at the first array and the other at the second array.

Recursive: Approach

Base case:

- If the a1 is empty but a2 has values -> return 1 // like the example
- If both a1 and a2 are empty; also return 1
- If a1 has values, but a2 is empty return 0

Recursive Case:

Use Pattern Matching?

| head a1 == head a2

choose to use the first element of a1, move to the next element of both arrays

choose to skip that first element of a1, move to the next element of just a2

add the counts of both to get the total count of ways

Handle the case where either one becomes empty

Compare the head values of each

python implementation:

```
def count_occurrences(a1, a2):
    if not a1 and a2:
        return 1

    if not (a1 and a2):
        return 1

    if a1 and not a2:
        return 0

    # front element of each array
    if a1[0] == a2[0]:
        # If the first elements match, we have two options:
        # 1. Use the first element of a1 and move to the next element in both lists.
        # 2. Skip the first element of a1 and move to the next element in a2.
        return count_occurrences(a1[1:], a2[1:]) + count_occurrences(a1, a2[1:])
```

```
    else:
        # If the first elements don't match, skip the first element of a2.
        return count_occurrences(a1, a2[1:])
-}
```

Solution:

```
count_occurrences :: [Int] -> [Int] -> Int
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (head1:tail1) (head2:tail2)
    | head1 == head2 = count_occurrences tail1 tail2 + count_occurrences (head1:tail1) tail2
    | otherwise = count_occurrences (head1:tail1) tail2
```